Microcode optimization for the PCS processor

François Bodin, François Charot, Charles Wagner

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1050

Programme 2

MICROCODE OPTIMIZATION FOR THE PCS PROCESSOR

François BODIN
François CHAROT
Charles WAGNER

Juin 1989

RR_1050

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Microcode optimization for the PCS processor[1]

François Bodin
François Charot
Charles Wagner

Mai 1989
26 pages
Publication interne n° 473

Abstract

This paper aims at describing a high-performance programmable pipeline architecture consisting of a linear array of PCS processors. The PCS processor which is capable of performing 20 million floating-point operations per second (20 MFLOPS) has been built from off-the-shelf chips on a wire-wrapped board. The prototype processor is attached to a SUN-3 workstation.

Efficient microcode is generated using the microcode compiler that has been designed and implemented. The microcode optimization includes microcode compaction and loop optimization using two techniques: software pipelining and loop unrolling. Preliminary results obtained on vector benchmarks are given.

Optimisation de microcode pour le processeur PCS

Résumé

Ce rapport décrit l'architecture d'un opérateur modulaire (PCS) réalisé à partir de composants discrets rapides et capable d'exécuter jusqu'à 20 millions d'opérations flottantes par seconde (20 MFLOPS). Ces facilités d'interconnexion permettent la mise en œuvre de structures parallèles performantes capables de supporter une parallélisation de type pipeline. Un processeur PCS prototype (carte réalisée en technologie "wrapping") est attaché à une station hôte SUN-3.

Le processeur PCS est composé d'unités fonctionnelles pipelines pouvant fonctionner en parallèle et de façon synchrone. La gestion de ce parallélisme interne est pris en charge par un compilateur de microcode. Celui-ci génère automatiquement, à partir du langage de haut niveau qui a été spécifié, un microcode efficace. Dans ce rapport, les composantes du compilateur sont successivement décrites. Le module d'optimisation du microcode (compaction de code, optimisation des boucles) ainsi que le modèle de machine qui rend le module d'optimisation indépendant de l'architecture sont plus largement décrits. Des résultats préliminaires sont donnés.

[1]La réalisation du processeur PCS a été menée en collaboration avec la société SOGITEC de Bruz avec le soutien de la Collaboration Bretagne Image (CBI) et a bénéficié d'aides de la Région de Bretagne et de l'Etat (DRIR).

1 Introduction

Technology offers us opportunities to develop parallel computation for both special-purpose and general-purpose devices. Among several approaches to parallel organization that can take advantage of these new possibilities, the systolic array concept is particularly interesting. A systolic array is a parallel device made of a number of identical processing elements which are regularly and locally connected; there is no link between two distant elements. In a systolic array, data circulate in the network in a pipeline fashion; the array makes multiple use of each input data. For compute-bound problems such matrix computation, this removes the I/O bottleneck between processor and memory. A systolic array usually operates synchronously; the flow of data through the array is organized in such a way that a processing element never has to wait for data and can consequently compute at its maximum speed. Over the past few years, the systolic array concept has materialized by the realization of systolic machines. Among the systolic arrays realized so far, there are prototype and industrial machines. MICSMACS [14] is a prototype VLSI programmable systolic array, which has been designed at IRISA. MICSMACS consists of a linear systolic array comprised of 18 full-custom VLSI 2 μm CMOS chips (the MICS module), and an interface board (the MACS module). The operation of the array is SIMD, each processor receiving a 16-bit instruction word every 100 ns. MICSMACS is connected to an IBM PC/AT microcomputer. The Warp machine [5] designed at Carnegie-Mellon University and jointly built with its industrial partners, is a linear systolic array of 10 identical cells, each of which is a 10 MFLOPS programmable processor. The Warp machine is currently produced and marketed by General Electric.

The PCS processor [7] we described in this paper aims at similar goals to Warp, even if its internal architecture organization is different. It has been designed as a high-performance programmable elementary processor of a linear array of cells to be used as a parallel hardware accelerator. This accelerator is able to support a wide class of applications in which the parallel implementation is based on pipelining since the use of FIFOs to support communications between processors allows the data exchanges between processors to be completely asynchronous. The PCS processor is implemented from off-the-shelf chips on a wire-wrapped board and its development is carried out with the SOGITEC company.

Efficient code is produced for the PCS processor using a microcode compiler. The microcode compiler exploits the low level fine grain parallelism inherent to the wide full instruction word concurrency of the PCS cell horizontal architecture. The optimization part includes microcode compaction and software pipelining. Microcode compaction builds efficient microinstructions by scheduling microoperations to execute with the highest possible concurrency consistant with parallelism inherent to the instruction word and data dependency considerations in the program. Software pipelining schedules looping operations in order to minimize the interval at which successive iterations are initiated and thus to have at any time multiple iterations of a loop in progress simultaneously in different stage of the computation. In order to be able to support various architectures made of multiple pipeline functional units, an important part of the work has been devoted to machine modeling that allows the microarchitecture of the cell to be fully specified and makes the compiler to be retargeted to architectures composed of multiple pipeline functional units quite straigthforward.

In this paper, in section 2 we first present an overview of the PCS-based system. The next sections deal with the compilation aspects of the PCS-based system. Section 3 is devoted to the PCS cell compiler organization. In section 4, the optimization part of the compilation process is presented. The micromachine modeling that allows the microarchitecture of the

cell to be expressed is also outlined. The techniques used to produce an efficient microcode (compaction and loop scheduling) are also described. Finally, preliminary results obtained on vector benchmarks (Livermore Loops, ..) are discussed in section 5.

2 Overview of the PCS Machine

A PCS cell-based hardware accelerator is fully exploitable if it is efficiently integrated in an existing hardware environment, namely a workstation. The access to the PCS-based system from the workstation has to be realized through an interface processor *(IP)*. IP is in charge of sending data to or receiving data from the leftmost and the rightmost cells of the array of PCS processors and managing the communications with the Host workstation. IP is of major importance in the system since it has to support all of its tasks in parallel. Even so, the global performances of the system will depend mainly on the bandwith between the host workstation and the array of PCS cells. It is obvious that the array of cells will be used especially to perform compute-bound tasks; tasks where the number of operations will be an order of magnitude larger that the I/O.

The components of the PCS machine as depicted in Figure 1 are:

- the host workstation
- the parallel hardware accelerator made up of an array of PCS cells (the prototype version only includes one PCS cell)
- the interface processor (IP)

The components of the PCS prototype machine will now be described. The way all the components are linked together and the communication mechanisms will also be outlined.

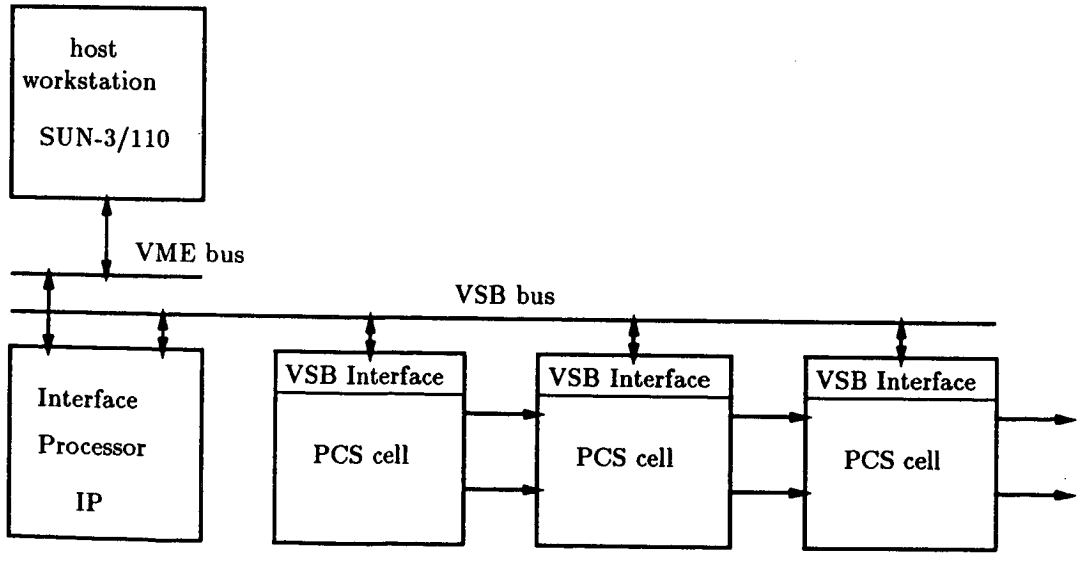


Figure 1 : The whole machine

2.1 The Host Workstation

The Host is a SUN-3/110 3-slot card cage workstation. The CPU board includes the 16.67-MHz MC68020 central processor, a MC68881 floating-point coprocessor, 4 Mbytes of main memory and a full 32-bit VMEbus interface. The VMEbus is dual-ported with master and slave interface; even so only the master mode has been considered in this application. The master interface provides access from the CPU to the VMEbus. The second slot is used for memory expansion (4Mbytes). The third slot receives the interface processor that is connected on the VMEbus.

Library routines have been written and are at users disposal. The PCS library includes functions to access the accelerator as well as system calls to control the execution of tasks on the accelerator. The way these routines are used is quite standard since they can be called like usual library functions in a C program. The execution of a routine is based on requests sent by the host to the interface processor.

2.2 The Interface Processor

The IP is a MC68020-based VME/VSB[2] processor board. The board includes 1 Mbyte of main memory with dual porting to the MC68020 processor and the VMEbus. This allows normal MC68020 processing and local bus activity to continue during access from the VMEbus. The board also includes a full VSB interface which allows different operations to be dedicated to either the VSB or VMEbus, optimizing processor and bus utilization.

Communications between the host workstation and the IP are supported via a shared memory implanted on the VMEbus. Whereas communications between the IP and the accelerator operate on the VSB. The communication mechanism between the host and the IP is based on interrupts. An interrupt is supplied via the VMEbus by writing at a given address in the shared memory.

The execution by the host of a library routine generates a request sent to the IP via the shared memory. Then the request, handled by the IP, activates a task on the interface processor. The tasks deal with the flow of data between the components, and the execution of treatments on the accelerator. These are mainly:

- in normal mode, the access to the input/output FIFOs of the accelerator, the local memory banks of PCS and the execution of tasks on the accelerator.
- in diagnostic mode, the control of the execution, the display of internal registers of PCS, etc.

2.3 The Array of PCS Cells

The PCS processor is the elementary operator of a linear array of cells to allow pipeline systems to be designed. The array of cells is attached to the host workstation by means of the interface described previously and constitutes the workstation parallel accelerator. The PCS cells are linearly connected and the data flow is unidirectional as illustrated in Figure 1. Each cell has two 32-bit input and output channels and can transfer up to 40 million 32-bit words to and from its neighbouring cells per second (160 Mbytes/second). The cell executes single-precision floating-point operations [3] at a peak rate of 20 MFLOPS.

[2]VSB stands for VME Subsystem Bus.

[3]double-precision floating-point operation is not supported by the architecture.

The architecture of the cell has been designed using high performance off-the-shelf chips and is composed of two parts: the operative part with its data path and the control part. The operative part and its main functions are depicted in Figure 2. All data paths are 32-bits wide.

The operative part of the PCS cell encompasses:

- a 32-bit floating-point multiplier/divider (*MPY*), implemented by a single component (the ADSP-3212), which supports 32-bit IEEE single-precision floating-point multiplications and divisions, as well as 32-bit two-complement fixed-point multiplications.

- a 32-bit floating-point ALU (*ALU*), implemented by the ADSP-3222 chip, which supports 32-bit IEEE single-precision floating-point addition, substraction, extended-precision integer operations, logical operations, data format conversions, floating-point division and square root are also included.

- two multiport register files (*RFA, RFB*) of 128 32-bit words, implemented by the ADSP-3128 chips are used to store intermediate results or scalar variables and to support the peak execution rate of the two computation units. Each register file has a bidirectional port used to interface with the memory banks.

- two memory banks (*MBA, MBB*) for local data storage. Each bank consists of 64K of 32-bit words and has its own arbitration logic which allows local access or external bus access to be performed.

- two address generators (*AGA, AGB*) for efficient memory management, implemented by the ADSP-1410 chips. Each generator is able to generate a 16-bit memory address, and to modify this memory address in a single instruction cycle. An address crossbar switch (*ACS*) allows external data to be provided via the input port of the address generator selected. This external data can come either from the input queues (16-bit least significant word), from the register files (16-bit least significant word) or from the microcode memory.

- two input FIFO queues (*FQA, FQB*). Each queue consists of 512 36-bit words (32-bit data and 4-bit condition) and are used to interface the cell with the neighbouring cells.

- a data crossbar switch (*DCS*), which links up the elements *FQA, FQB, RFA, RFB, MPY, ALU*; the computation units and the fast storage units are tightly linked together. This choice has the advantage of allowing the chaining of arithmetic operations and therefore reduces the use of the multiport register files. This network has six input ports and eight output ports and it can be reconfigured in every cycle under microcode control.

The floating-point units are seen as 3-stage pipeline units and all the operations except division and square root can be initiated every instruction cycle. The PCS cell is horizontally microcode controlled. The cell has a 16K words of microcode memory and its own microsequencer implemented by the Am29331 chip. Such a large microcode memory suggests that entire application programs can be loaded into the cell avoiding the need to reload PCS during the processing. The microcode is loaded through a specialized interface.

The use of the PCS cell must be as flexible as possible; we want to be able to associate a particular treatment to a given input data. For that reason, we have chosen to associate 4 bits called synchronization conditions (SC) with each input and output datum. These SC are tested by the microsequencer in order to generate the right treatment associated with the input data.

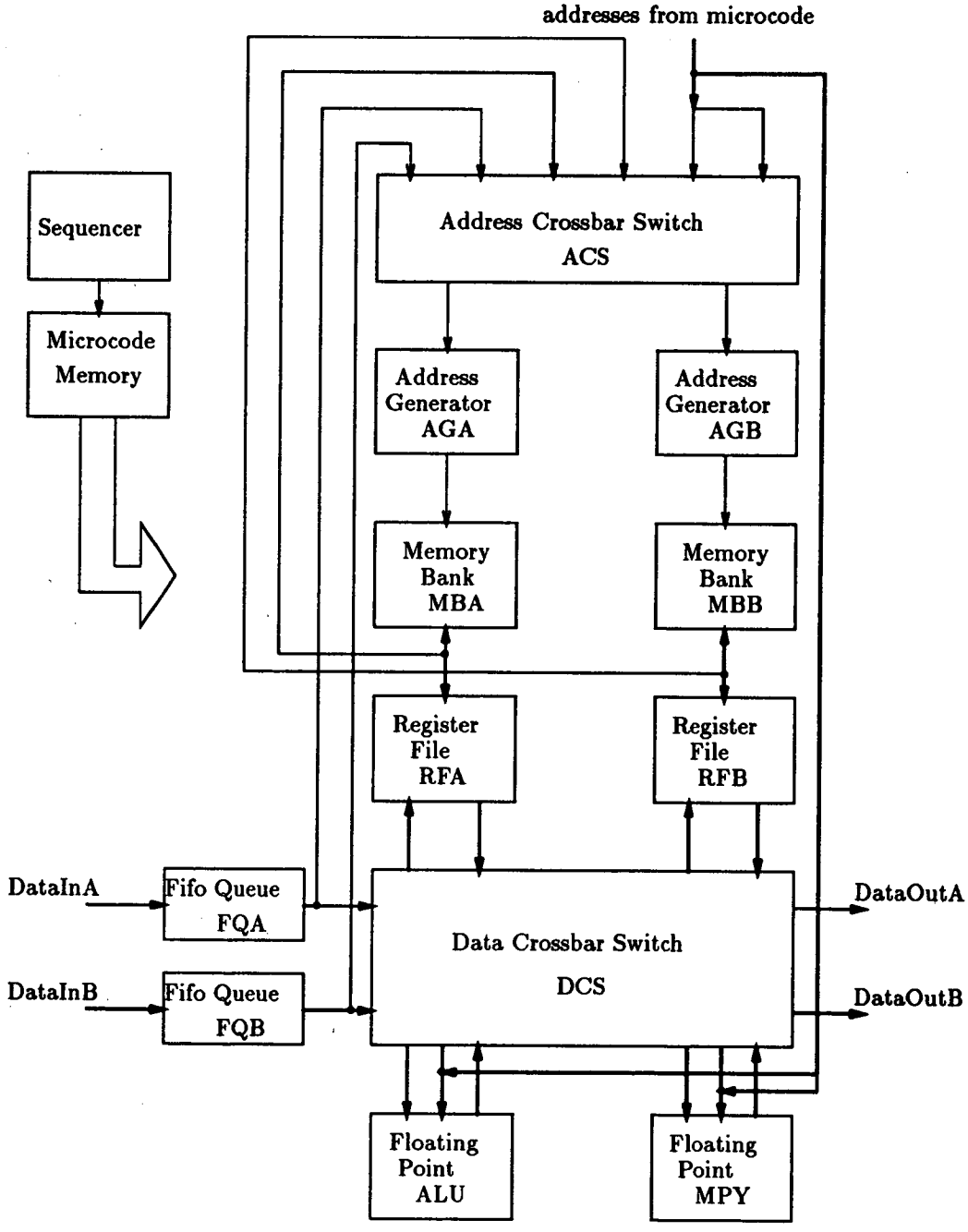


Figure 2 : PCS cell data path

In order to implement an algorithm in a pipeline way, a synchronization mechanism between cells is provided. Several cells can be easily connected and form a linear pipeline system. But each cell in the array has to execute a particular code as is the case in a MIMD machine. The data flow control is made step by step. The implementation of such an algorithm raises the problem of the transmission of SC through the pipeline. A cell that transmits data to another cell must generate SC to indicate the treatment to be performed by this cell. PCS supports three modes of SC generation: either four bits issued from the microinstruction are passed to the neighbour cell, or the input SC are sent to a finite state machine also controlled by a microinstruction field, in such a case, the SC is generated by the finite state machine, or the last mode called the transmission mode, where the SCs received by the cell are directly sent to the next cell without modification.

3 Compilation for a PCS-Based Architecture

This section is devoted to the compilation aspects of the PCS-based architecture. The **lpcs** language used to program the cell is first presented, then a brief overview of the compiler organization is given.

3.1 The lpcs Language

The **lpcs** is the language used to program the PCS cells. We suppose that the algorithm to be mapped on the array of cells has been previously decomposed into pipeline or parallel tasks which communicate locally. These tasks can then be expressed using the **lpcs** language. This language was defined to simplify cell programming; the microcode level is hidden from the user. However, the language was restricted to constructs that run efficiently on the cells. This choice was made because of the specialization of the machine to pipeline and/or systolic mode of operation. In the systolic mode, each cell executes the same **lpcs** program. In the pipeline mode, a different **lpcs** program is written for each cell.

The **lpcs** language is a block structured language with assignment, conditional, loop and procedure statements. The main particularity of **lpcs** is the communication primitives which allow inter-cell data exchanges to be specified. Communications between cells are specified explicitly with the asynchronous communication primitives **in** and **out**. Moreover, cells can only communicate with their neighbours. The **in** primitive receives data from the left cell, or from the host if the cell is the first of the array. The **out** primitive sends data to the right cell or to the host if the cell is the last one. Though the internal parallelism of the architecture is managed by the compiler, the inter-cell communication must be checked by the user. This can be done using the PCS simulation tools, and in practice this is very simple, since all communications can be controlled by the software. Moreover, the user has access to the FIFO flags (full, half-full and empty flags) in the language.

The abstraction level provided by the language allows the cell to be very simply programmed. For example, a complex algorithm such as the parallel geometric operations in the ray-tracing algorithm has been straightforwardly programmed on the array of cells.

Figure 3 shows a simple example of a **lpcs** program which evaluates a 4×4 matrix multiplication ($C = A \times B$) using an array of four cells. The program is composed of two parts. The first one initializes the data for the cell with the A matrix. The second one reads the B matrix elements in FIFO 1 and computes the partial dot product of elements of the C matrix which are sent in FIFO 2 of the neighbouring right processor.

```
program
{
  float Lig[4];
  int i,j;
  float x,y,row,resin,resout;
  for i= 1 to 4 do
    {
      in(1,x); Lig[i] = x;
      for j=2 to 4 do
       {
         in(1,y);out(1,y);
       }
      out(1,0);
    }
  for i = 1 to 4 do
    {
      in(1,row);
      for j = 1 to 3 do
       {
         in(1,x); out(1,x);
       }
      out(1,0);
     for j = 1 to 4  do
      {
        in(2, resin);
        resout = resin + row * Lig[j];
        out(2, resout);
      }
    }
}
```

Figure 3 : A **lpcs** program example

3.2 Overview of the cl Compiler

The main problem that is to be solved by the compiler is the management of the horizontal microcode parallelism of the PCS cell. The input to the compiler is a **lpcs** program, and the output consists of a horizontal microprogram for the cells. Figure 4 illustrates the compiler organization which is composed of four major modules:

- the *syntax analyzer* which translates the **lpcs** program into quadruple forms [1]. This phase is machine independent and can be easily updated to enhance the **lpcs** language.
- the *microcode generator* which translates quadruples into sequential microcode. This phase is machine dependent and manages resources like registers and memory banks.
- the *machine description parser* which parses the machine description and generates the input to the optimizer. The machine description allows all the features and timing constraints of the microarchitecture to be fully specified. This part is of major importance in order to be able to support any improvement and modification of the hardware and makes the optimizer completely machine independant.
- the *microcode optimizer* which parallelizes the sequential microcode. The optimizer is the most typical part of a compiler for a horizontal microarchitecture. It manages the multiple pipeline functional units and the fine grain parallelism of the machine.

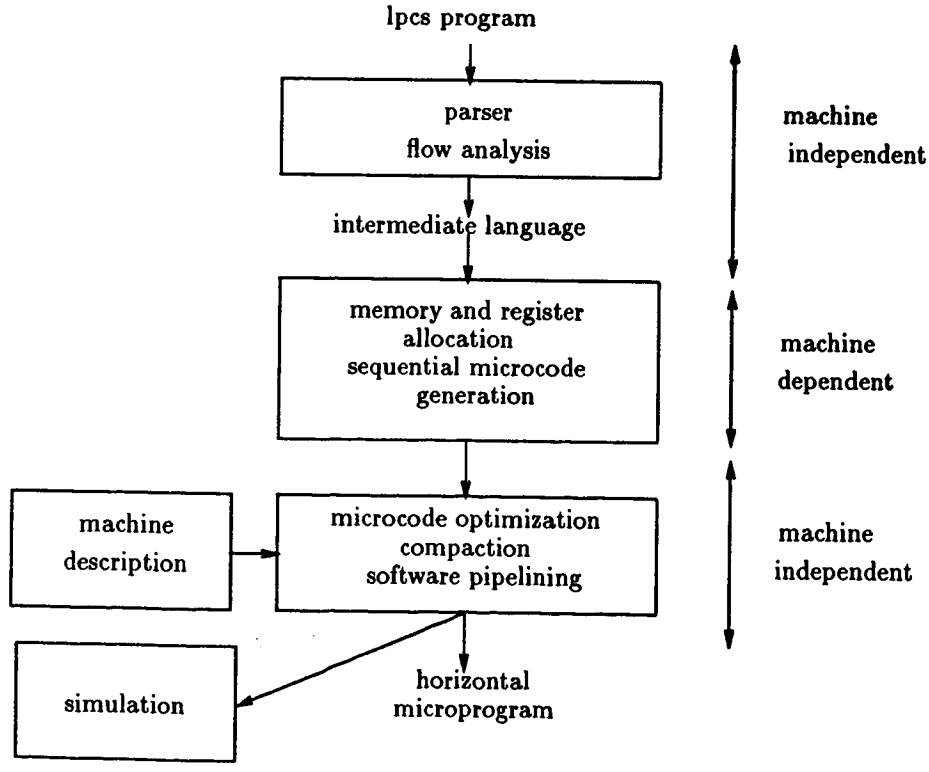


Figure 4 : Compiler organization

The microcode generator consists of many steps. The first one is the global flow analysis phase which collects information for all variables of the program. These information are used to allocate memory banks to variables and registers. To allocate registers we use the graph coloring method proposed by Chaitin [6]. The memory bank allocation is also done by a similar method. The sequential microcode is then generated by macro-expansion [22].

The sequential microcode and the machine description are input to the optimizer which generates a horizontal microcode for the target machine. *Loops* and *basic blocks* of the program are parallelized. The machine description used by the optimizer gives the behaviour of the machine and permits to calculate both the conflicts on the functional units and the data dependencies. For loops it uses global data dependencies (loop carried dependencies) which are calculated by the front-end of the compiler [4]. This computation is not done at high level statement but at the level of memory reference. This does not present more difficulties than building them at the lpcs statement level. The optimizer applies techniques like *microcode compaction* and *software pipelining*. The software pipelining is a loop optimization technique which minimizes the interval at which successive iterations are initiated, smaller the interval, higher the throughput.

4 Microcode Optimization

We first present the machine modeling which allows the complete description of the PCS cell architecture to be specified. The local compaction algorithm used to schedule microoperations is then described. Finally, the optimization phase which is the most important part of the compilation process is considered.

4.1 Micromachine Modeling

Machine modeling is of major importance in order to make the optimization process machine independant. Moreover, an important feature of machine modeling is that the behaviour of resources must be entirely synchronous and predictable. The model that has been defined is close to existent models [16] which have been enhanced to handle architecture features like:

- transient resources: a resource (for example a register) is said to be transient if its value is implicitly destroyed after some time, the value has to be used before it is lost.
- multi-cycles microoperations: some operations are not executed in one machine cycle. Therefore, an operation which uses a result produced by such a microoperation has to be delayed.
- delay branches: there are delay branches when the sequencer unit has pipeline stages.

Therefore, the model introduces temporal restrictions between microoperations, the solution to this problem is described in section 4.1.5 which is devoted to the compaction algorithm.

4.1.1 Resources

The machine resources are the functional units of the target machine. We distinguish four *types* of elements:

- **static** which stores a data permanently, except if there is a modification operation in the program.
- **latch** which can be read and written simultaneously.
- **transient** which stores the data only during the write cycle of the data.
- **field** used to code the microoperation.

The compiler gets all the features of the resources from a description which contains all the resources of the micromachine. For instance a PCS register bank declaration is:

```
STORAGE : RFA /* name of the register bank */
TYPE : STATIC /* resource type */
              /* resource capacity: 128 32-bit words */
CAPACITY : ( 32 , 128 )
ESTO
```

In the following section, R is the set of the resources of the micromachine and $Type$ is a function which gives the type of a resource:

$$Type : R \rightarrow \{transient, latch, static, field\}$$

4.1.2 Microoperation

The description must supply all the knowledge on the behaviour of the microoperations in order to calculate conflicts on resources and data dependencies. This is done by providing timing information for all the resources. A microoperation (MO_i) is defined by a sextuplet composed of the following fields:

$$<name, input, output, element, field, synchro>$$

- **name** identifies the microoperation.
- **input, output, element** describe the input resources I_{MO_i}, output resources O_{MO_i}, and elements of the architecture U_{MO_i} which are neither input nor output resources ($I_{MO_i} \subset R$, $O_{MO_i} \subset R$, $U_{MO_i} \subset R$).
- **field** is the set of fields and their values ($C_{MO_i} \subset R \times \mathbf{N}$) for the coding of the microoperation.
- **synchro** is a list of quadruples which describe the temporal use of the resource. They have the following structure:

$$<resource, begincycle, endcycle, R/W>$$

  - **resource** is the name of the resource and must be in another field of the description of the microoperation.
  - **begincycle** and **endcycle** specify the time at which the resource is used and released.

- the **R/W** attribute indicates how the resource is used (write or read). A resource which cannnot be shared is declared with the **W** attribute.

We define the function T_{MO_i}:

$$T_{MO_i} : I_{MO_i} \cup O_{MO_i} \cup U_{MO_i} \cup C_{MO_i} \rightarrow \mathbf{N} \times \mathbf{N} \times \{read, write, nil\}$$

which indicates how and at what time a resource is used when the microoperation is initiated.

In order to illustrate these notions, the declaration of an ALU microoperation is given in Figure 5.

1. the operands: $I_{MO_i} = \{aluina, aluinb\}$.
2. the result: $O_{MO_i} = \{aluout, AluCond\}$.
3. the internal pipeline stages:
   $U_{MO_i} = \{alu, alu1, alu2\}$.
4. the coding: $C_{MO_i} = \{(Alu\_Opcode, 2)\}$.
5. the way the resource is used:
   $T_{AluOperation}(aluina) = (0, 1, read)$
   $T_{AluOperation}(aluout) = (2, 3, write)$
   $T_{AluOperation}(Alu\_Opcode) = (0, 1, write)$
   ...

Figure 5 : Declaration of an ALU microoperation

4.1.3 Data Dependency

The first step of the optimization process consists of constructing the dependency graph. The data dependency is a partial order on microoperations which represents a set of precedence relation to be satisfied in order to preserve the original program semantic. To represent this data dependent relation (Δ), we use an acyclic directed graph (GD) whose nodes are microoperations. Let BB be a basic block of microoperations:

$$BB = MO_1, MO_2, ..., MO_n$$

Let $GD = (S, P)$ be the graph, S the set of vertices labeled with elements of BB and P the set of edges $(MO_i, MO_j) \in S \times S$. Let r be a resource of R. There is an edge between MO_i and MO_j if $MO_i \stackrel{\Delta}{\rightarrow} MO_j$, according to the following rules:

1. (MO_i **is direct dependent** on MO_j
   $MO_i \stackrel{\delta}{\rightarrow} MO_j$) if:

$$j > i,\ O_{MO_i} \cap I_{MO_j} \neq \emptyset$$
$$\nexists MO_i \stackrel{*\Delta}{\rightarrow} MO_k \stackrel{*\Delta}{\rightarrow} MO_j$$

2. MO_i is **output dependent** on MO_j ($MO_i \xrightarrow{\delta^o} MO_j$) if:

$$j > i,\ O_{MO_i} \cap O_{MO_j} = \{r\}$$
$$Type(r) = static$$

3. MO_i is **anti dependent** on MO_j ($MO_i \xrightarrow{\overline{\delta}} MO_j$) if:

$$j > i,\ I_{MO_i} \cap O_{MO_j} = \{r\}$$
$$Type(r) = static$$

4. $MO_i \xrightarrow{\Delta} MO_j$ if:

$$MO_i \xrightarrow{\delta} MO_j \text{ or } MO_i \xrightarrow{\overline{\delta}} MO_j \text{ or } MO_i \xrightarrow{\delta^o} MO_j$$

The set of transient and latch resources cannot introduce anti dependence or output dependence between microoperations.

4.1.4 Delay Between Microoperations

To be correct, the building of the microprogram must satisfy the delays between microoperations. To deal with delays, edges are valued with a timing pair:

$$(type, d) \subset \{\geq, =\} \times \mathbf{N}$$

this expresses the delay to be satisfied between the two microoperations that are linked by an edge.

If *type* is $\geq$, then the two microoperations must be distant of at least d cycles. If *type* is $=$, then the delay between the microoperations has to be equal to d and the delay is said to be a strict delay. The delay between two microoperations is evaluated depending upon the type of the resource involved in the data dependency and the type of the dependency. For instance, if MO_j depends on MO_i by a direct dependence ($MO_i \xrightarrow{\delta} MO_j$) and if MO_i is scheduled at cycle c and writes a transient resource r, n cycles later, then MO_j must be scheduled at cycle $c+n$.

The graph GD enhanced with all the delays is called GDD. All the delays are automatically deduced from the description of the microoperations and the resources. Let ω be the function that associates to MO_i the time at which it is initiated. A correct scheduling ω of the microoperations has to satisfy these relations:

- if $MO_i \xrightarrow{\geq d} MO_j$ then:

$$\omega(MO_j) - \omega(MO_i) \geq d$$

- if $MO_i \xrightarrow{=d} MO_j$ then:

$$\omega(MO_j) - \omega(MO_i) = d$$

4.1.5 Local Compaction Algorithm

The local compaction algorithm [9] schedules basic blocks of microoperations and tries to minimize the execution time. Dewitt has shown that the problem is NP-complete, because this problem is very similar to the resource constrained scheduling problem [15]. Three compaction algorithms are usually considered:

- the *list scheduling* algorithm which may use a First Come First Serve priority [8]. The microoperations are examined in the original order of the microcode. Each microoperation is scheduled as soon as possible, and when a microoperation is scheduled it is never disallocated.
- the *critical path* algorithm defined by Kleir and Ramamoorthy [18] which identifies critical microoperations and tries to schedule the other without extending the scheduling.
- the *branch and bound* algorithm which guarantees the optimality of the solution since all the solutions are examined, however, the complexity is exponential.

In practice, heuristic algorithms give generally near optimal results. To be correct the algorithm must satisfy the following two conditions:

- *resource constraints*: the scheduling may not introduce conflicts between the functional units of the machine. To avoid conflicts on resources, we use the reservation table formalism [19].
- *Data dependency and delay.*

The compaction algorithm is used to schedule the basic blocks of the microprogram and is a part of the loop scheduling algorithm. The main problem of micromachine modeling is the handling of strict delays, that have edges valued with $=$. To deal with this problem, we introduce the notion of **template** which is a set of microoperations connected by strict delays. Two microoperations belong to the same template if there exists a path between the microoperations in GDD and if all the edges along the path are valued with strict delays.

By definition, all the templates are disjoint. There exists a single scheduling of the templates since all the microoperations are linked by strict delays. This notion allows the microoperations to be scheduled without the use of a backtracking algorithm. The following condition ensures that the scheduling of the templates exists:

Compaction condition: Let $GB = (EB, AB)$ be the template graph whose nodes are templates. This graph is deduced from the GDD graph. If GB is acyclic and if all the templates are legal then there exists a correct scheduling ω of the microoperations.

Proof: The proof is based on the use of a list scheduling algorithm on the graph of templates, and the scheduling of microoperations within a template.

Let $LB = B_1, ..., B_i, ..., B_n$ be the HLF (High Level First) list of templates. A template B_i is legal if there exists a scheduling ω_{B_i} that respects delays and data dependencies between the microoperations of the template without conflict on resources.

We choose the list scheduling algorithm with the HLF priority. The scheduling $\omega_B : LB \rightarrow \mathbf{N}$ is done on the template first. The algorithm is the following:

- Input: the template graph GB.

- Output: the scheduling of the template ω_B.

  1. $cc = 0$;
  2. **While** $LB \neq \emptyset$
     (a) **For** each template B in LB **Do**
     If B can "be allocated" at cc
     Then schedule B, update the reservation table and LB.
     (b) $cc++$;

By "be allocated", we mean that data dependencies and delays are satisfied and there is no conflict on the resources.

The microoperation scheduling ω is then obtained by the function:

$$\omega(MO) = \omega_B(MO \in B_i) + \omega_{B_i}(MO)$$

4.2 Loop Optimization

Loop optimization is important in order to produce an efficient microcode. It is brought to the fore with the following example which adds two data read from the input queues and then sends the result to the PCS cell output register:

```
for i=1 to 10 do
    read fifo A,read fifo B;
    add;
    write output A
enddo
```

Based on the timing constraints of the architecture, the execution of one iteration is the following:

```
cycle c    : read fifo A, read fifo B
      c+1  : nop
      c+2  : add
      c+3  : nop
      c+4  : nop
      c+5  : write output A
```

Without parallelism between iterations the loop takes 60 cycles to complete. However, an iteration can be initiated every cycle and its execution takes then 15 cycles as illustrated by the following scheme:

```
cycle c    : read fifo A,B
      c+1  : nop            read fifo A,B
      c+2  : add            nop           read fifo A,B
      c+3  : nop            add           nop
      c+4  : nop            nop           add
      c+5  : write output A nop           nop
      c+6  :                write output A nop
      c+7  :                              write output A
```

The parallelization of loops has been approached in different ways. The methods proposed by Touzeau [25] and Eisenbeis [10] are an extension of techniques used to optimize static hardware pipelines [23]. Other algorithms are based on global scheduling algorithms like Trace scheduling [13] or Percolation scheduling [2]. Their principle is first to unroll the loop and then to compact it. The main advantage of such methods is their ability to handle complex loops with branching. However, they do not take into account the cyclic regularity of the loop, and therefore generate great microcode size.

In this section, the loop optimization algorithm used in the compiler is described. It uses techniques defined in [10] [25]. We focus our attention on optmizing simple loops; loops made up of assignment statements without branching.

4.2.1 Data Dependency, V-Loop

The scheduling of a loop must deal with two types of data dependencies:

- the *local data dependencies* (intra-iteration) of the body of the loop (GD).
- the *global data dependencies* of the loop (loop carried dependencies) which give the precedence to the constraints between the microoperations of successive iterations.

The scheduling of a loop is correct only if it respects the two types of dependencies. For instance in a loop containing the statement $a = a + A[i] * B[i]$ a given iteration cannot read the value of a before it has been written by the previous iteration because of the direct data dependency on variable a.

In order to take into account loop carried dependencies the loop body graph GD is enhanced with the global dependencies. The graph is now called GGD. If this graph is acyclic the loop is said to be a vector loop (V-loop [11]) otherwise it is a recurrent loop (R-loop).

4.2.2 Software Pipelining

The algorithm searches first the scheduling of one iteration, then the final loop is pipelined based on this scheduling. In this paragraph we are only interested in V-loop. The R-loop problem is described in the next paragraph. The algorithm searches a scheduling which has the following two properties:

- all the iterations have the same scheduling.
- the latency (*initiation interval*) between two successive iterations is L.

This problem can be resumed in searching a global scheduling of the loop Ω that satisfies the data dependencies and is deduced from a local scheduling of the loop body:

$$\Omega(MO_i^j) = j * L + \omega(MO_i)$$

with $MO_i^j, 1 \leq i \leq m, 1 \leq j \leq N$ the microoperation MO_i of iteration j. For a V-loop, a global scheduling of this form satisfies the global dependencies for all positive values of L. With the global scheduling we can construct a new loop, semantically equivalent to the original one. This is called a software pipelining. Figure 6 illustrates the software pipelining execution where the loop body is divided into 3 parts (C_1, C_2, C_3) concurrently executed. The algorithm consists in two parts:

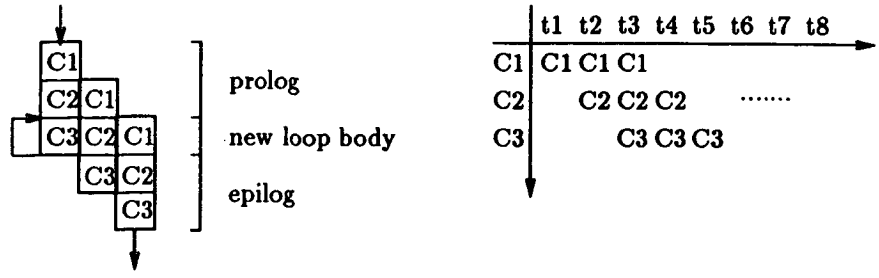


Figure 6 : Software pipelining

- Find a scheduling ω. This scheduling must be compatible with the minimum latency L and has to satisfy the data dependencies and the use of resources. In Figure 7, microoperations MO_1, MO_2, MO_3, MO_4 are executed in parallel since successive iterations are initiated with latency L, the amount of resources used by these microoperations may not exceed the machine resources.

- Verify that latency L is compatible with register occupation If not, there are two solutions. The first one is to try another latency $L + 1$, the second one is to apply technique like modulo variable expansion [20] or the method defined by Eisenbeis et al [12].

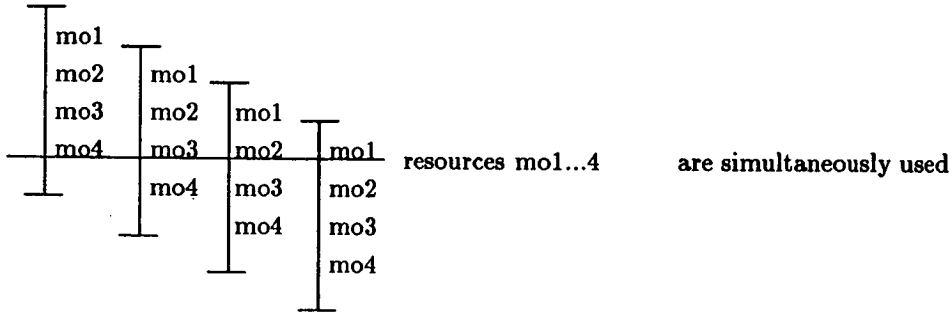


Figure 7 : Resource constraints

4.2.3 Loop Body Scheduling

The algorithm used to find the scheduling of the loop body is similar to the local compaction algorithm, except that:

- The global data dependencies graph *GGD* is used. It guaranties that loop carried dependencies are respected for all latency values.

- We apply the modulo constraint [24] for latency L on the resource to ensure that the L latency does not generate resource conflict.

Modulo constraint: let ω be a scheduling of the loop body C. It is compatible with the *modulo constraint* for L if:

$$\begin{aligned} \forall MO \in C, \forall r \in R \mid T_{MO}(r) = (i,j,r/w) \\ \textbf{then } \nexists MO' \in C, k \in \mathbf{Z} \\ \textbf{with } T_{MO'}(r) = (i',j',r/w) \\ \textbf{and } [\omega(MO)+i+kL, \omega(MO)+j+kL[\\ \cap [\omega(MO')+i', \omega(MO')+j'[\neq \emptyset \end{aligned}$$

The problem is to limit the latency search space. However, we can restrict this search to an interval defined by the following conditions:

- The lower bound is the minimum number of cycles imposed by the critical resources. A resource is said to be critical if the removal of the contribution of the resource in the lower bound calculation reduces the lower bound.
- The higher bound is the length of the scheduling of the loop body without modulo constraint. If this value is reached there is no parallelism between iterations.

4.2.4 Loop Optimization Based On Loop Unrolling

In paragraph 4.2.2, we have shown how the optimization of V-loops is treated. We sketch here another loop optimization algorithm which deals with R-loops in a more natural way than the software pipeling technique does. The basic idea of the algorithm is to search a period in an unrolling/compaction process. The compaction algorithm used is quite similar to the one defined previously; when a period is found in the scheduling of the iterations it is used to construct a software pipelining of a loop. In order to illustrate the method, let us consider the following loop:

```
for i = 1 to 100 do
{
 D[i] = C[i] + A[i] * B[i];
}
```

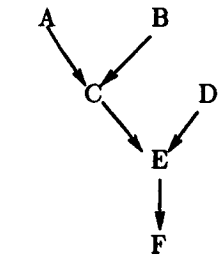
Let C be the body of the loop. The algorithm uses the unrolling function u:

$$u^i(C) = C_1 C_2 ... C_i$$

where i is the degree of the unrolling and C_j are the successive occurrences of the loop body. For instance, the $u^3(C)$ of the previous loop is:

```
D[i] = C[i] + A[i] * B[i];
D[i+1] = C[i+1] + A[i+1] * B[i+1];
D[i+2] = C[i+2] + A[i+2] * B[i+2];
```

If we unroll and compact successively several iterations of the loop, then we always obtain the same scheduling of the iterations after an unrolling of degree 2, as shown in Figure 8 where successive iterations are listed separatly, side by side. The vertical axis is time; all statements on a horizontal line are simultaneously executed. We suppose that only two concurrent memory



the dependency graph

	1	2	3	4
1	A B			
2	C D	A		
3	E	B D		
4	F	C	A	
5		E	B D	
6		F	C	A
7			E	B D
8			F	C
9				E
10				F

code atfer 4 phases

Figure 8 : Example of loop unrolling

access (A, B, D, F in the figure) are authorized without conflict, and that all operations take one cycle to complete. Then, we can construct a new loop with two cycles latency between successive iterations.

Aiken and Nicolau [3] have proposed another approach which also uses a greedy scheduling and unrolling algorithm. Their algorithm gives an optimal scheduling, but only consider data dependencies. If resources are considered the problem is NP-complete. The algorithm we proposed considers resources but it does not necessarily give an optimal scheduling.

The interesting idea of this algorithm is the use of the unrolling to find a cyclic scheduling of the loop. The algorithm permits to construct a software pipelining which is equivalent to a complete unrolling of the loop except for the first iterations. The other idea of the algorithm is to extend the space of solutions for the scheduling of loops on the basis of classical software pipelining techniques. The software pipelining technique asserts that all the iterations have the same scheduling, our algorithm enables several iterations to have different schedulings.

5 Preliminary Results

The performances of the PCS cell are discussed in this section. We present two sets of benchmarks.The first one is the 14-loop Livermore kernel [21] which has been chosen as program fragments representative of scientific programs. The second one is the MVF vector library kernel [11] which is given in Table 1. These results have been obtained on the prototype cell which so far does not take into account some hardware improvements. Subtantial performance improvements can be expected in the near future.

The performances of loops using software pipelining are analysed using the Hockney formula [17]:

$$T(N) = \frac{(N + N_{1/2})}{V_\infty}$$

- N: is the number of iterations of the loop.

m1	y(i) = (a * x(i)) + b
m2	y(i) = (a + x(i)) * b
mv1	z(i) = ((x(i) * y(i)) + a) * b
mv2	z(i) = ((x(i) * b) + a) * y(i)
mv3	z(i) = (y(i) + x(i)*a) + b
mv4	z(i) = (x(i) * a) + (y(i)) + b)
mv5	z(i) = ((a *x(i)) + b) + y(i)
mv6	z(i) = ((x(i) + b) * a) + y(i)
mv7	z(i) = ((x(i) + y(i)) * a) + b
mv8	z(i) = ((x(i) + y(i)) + a) * b
mv9	z(i) = ((x(i) + a) + y(i)) * b
mvf1	t(i) = ((x(i) * a) + y(i)) * z(i)
mvf2	t(i) = ((x(i) + y(i)) * z(i)) + a
mvf3	t(i) = (x(i) + (y(i) * z(i))) * a
mvf4	t(i) = x(i) + ((y(i) * a) * z(i))
mvf5	t(i) = (x(i) + y(i)) + (x(i) * a)
mvf6	t(i) = (x(i) + (y(i) * a)) + z(i)
mvf7	t(i) = (x(i) * y(i) + a) * z(i)
mvf8	t(i) = ((x(i) +a) * y(i)) * z(i)
mvf9	t(i) = (x(i) + y(i)) *(a + z(i))

Table 1 : MVF kernel

- $T(N)$: is the execution time for N iterations.
- V_∞: is the asymptotic speed (in MITS: Million ITerations per Second), which is the speed obtained when N is the infinity.
- $N_{1/2}$: is the number of iterations to get half of V_∞ ($V_{1/2}$).
- $v(N)$: is the speed in MITS for N and is equal to:

$$v(N) = \frac{N}{T(N)}$$

The performances of the 14-loop Livermore kernel are given in Table 2 and are illustrated with the $N_{1/2}$ value in column 2 (number of iterations to get half the value of V_∞) and with the asymptotic speed in MFLOPS in column 3. The Fortran programs were manually translated into the **fpcs** syntax. The translation was straightforward. Kernel 8 was simply compacted because of the great number of operations (more than 300). For multiple loops, only the internal loop is considered.

Table 3 shows results for the MVF vector library kernel. All the performances are optimal for this set of loops. The goal of the m, mv, mvf families is to get a systematic evaluation of the behaviour of the different code generators for loop operations. The loop *mul comp* is the multiplication of two complex vectors and *drot* a Givens rotation.

The data dependency and the critical resource bottleneck are the main factors of the limit of the maximum achievable rate. In the case of the prototype PCS cell the critical resource

Loop	$N_{1/2}$	V_∞
kernel 1	4.2	10
kernel 2	4.4	10
kernel 3	1.7	5
kernel 4	1.7	5
kernel 5	1.9	2
kernel 6	4.3	6.7
kernel 7	3	8.8
kernel 8	0	4.5
kernel 9	2.7	7.7
kernel 10	2.9	2.5
kernel 11	1.8	1.6
kernel 12	4.4	4
kernel 13	3.1	2.5
kernel 14	6.9	4.2
average	3.3	5.6

Table 2 : Livermore kernel

Loop	$N_{1/2}$	V_∞
m1	4	10
m2	4	10
mv1	4	10
mv2	4	10
mv3	4	10
mv4	3	10
mv5	4	10
mv6	4	10
mv7	4	10
mv8	4	10
mv9	4	10
mvf1	4	10
mvf2	4	10
mvf3	4	10
mvf4	4	10
mvf5	3	10
mvf6	4	10
mvf7	4	10
mvf8	4	10
mvf9	3	10
mul comp	2	10
drot	1.8	10

Table 3 : MVF kernel

bottleneck is mainly due to two resources: the ALU functional unit and register access. The ALU functional unit constitutes a resource bottleneck when address calculations are done to it, because of the address generator saturation. This is the case in kernel 8. Register access constitutes the other most frequent bottleneck and this is the case when all operands come from memory banks because in this case chaining optimization is limited. The other limit is data dependency. However, the small number of pipeline stages permits to obtain quite acceptable performances in case of recurrences as shown in the following *dot product* loop:

```
for k = 1 to 1024 do
  {
       q = q + x[k] * z[k];
  }
```

which runs at 5 MFLOPS.

6 Conclusion

We have described a prototype pipeline machine made of a linear array of identical PCS processors. The PCS processor is implemented using off-the-shelf high-performance chips on a wire-wrapped board and is controlled by a standard MC68020-based VME/VSB processor board interface that sends control functions and handles inputs and outputs. The interface board is attached through the VMEbus interface to a SUN-3 host workstation. The machine is a testbed for image synthesis techniques and all kinds of compute-bound algorithms. The current application that has been tested on the machine is the geometric operations of an image synthesis by polygons algorithm.

We designed and implemented a microcode compiler for the PCS processor. Once the algorithm to be mapped on the architecture has been decomposed into pipeline and/or parallel tasks (the level of decomposition is taken into account by the user), the microcode compiler is capable to generate an efficient microcode for each task and all the communications between tasks are asynchronous. The compiler manages the low level fine grain parallelism inherent to the wide full instruction word concurrency of the PCS horizontal architecture. The optimization part of the compiler includes microcode compaction and loop optimization using a software pipelining technique. Another loop optimization method based on the unrolling of the loop is under investigation and the sketch of the algorithm has been given. The comparison of these two loop optimization techniques will be completely described in a subsequent paper.

Acknowledgements

We thank Bernard Chardevel and Frank Rousée from the SOGITEC company for their contribution towards the construction of the wire-wrapped prototype PCS board. We are also grateful to William Jalby for his valuable comments and advice on earlier versions of this paper.

References

[1] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[2] A. Aiken and A. Nicolau. A development for horizontal microcode programs. *MICRO 19*, 23–31, 1986.

[3] A. Aiken and A. Nicolau. Optimal loop parallelization. *Proceeding of the SIGPLAN '88*, 308–317, 1988.

[4] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Transaction on Programming Languages and Systems*, 9(4):491–542, 1987.

[5] M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, and J. Webb. The Warp Computer: Architecture, Implementation and Performance. In *IEEE Tr. on Computers*, December 1987.

[6] G.J. Chaitin. Register allocation and spilling via graph coloring. *ACM*, ():98–105, 1982.

[7] F. Charot and F. Rousée. CSI: A Processor for Image Synthesis. In *EUROGRAPHICS'86 Conference Proceeding*, September 1986.

[8] S. Dasgupta and J. Tartar. The identification of maximal parallelism in straight-line microprograms. *IEEE Transactions on Computers*, 25(10):986–991, 1976.

[9] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett. Local microcode compaction techniques. *Computing Survey*, 12(3):261–294, 1980.

[10] C. Eisenbeis. Optimisation automatique de programmes sur array-processors. Thèse d'université de Pierre et Marie Curie Paris 6, Juin 1986.

[11] C. Eisenbeis. Optimization of horizontal microcode generation for loop structures. *ACM Supercomputing 88*, 453–465, 1988.

[12] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Squeezing more cpu performance out of a cray-2 by vector block scheduling. *Florida Supercomputing 88*, 1988.

[13] J.A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[14] P. Frison and Dominique Lavenier. A VLSI systolic machine for string correction. In *Third International Conference on Supercomputing*, Boston (U.S.A), May 1988.

[15] M. R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and company, 1979.

[16] J. L. Gieser. On horizontally microprogrammed microarchitecture description techniques. *IEEE Transactions on Software Engineering*, 8(5):513–525, 1982.

[17] R.W. Hockney and C.R. Jesshope. *Parallel Computers.* Adam Hilger Ltd, Bristol, 1981.

[18] R.L. Kleir and C.V. Ramamoorthy. Optimization strategy for microprograms. *IEEE Transactions on Computers*, 20(7):783–794, 1971.

[19] P. M. Kogge. *Architecture of Pipelined Computers.* Mc Graw-Hill, 1981.

[20] M. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, Carnegie Mellon University, May 1987.

[21] F.H. McMahon. *LLNL Fortrans Kernels: MFLOPS.* Mars 1984.

[22] M. Mezzalama and P. Prinetto. A machine-independent approach to microprogram synthesis. *Software-Practice and Experience*, 12:985–1010, 1982.

[23] J.H. Patel and E.S. Davidson. Improving the throughput by insertion of delays. *Proc 3rd Annual Symp. on Computer Architecture*, 159–164, 1976.

[24] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *IEEE*, 183–198, 1981.

[25] R.F. Touzeau. A fortran compiler for the fps-164 scientific computer. *Proc. of the ACM SIGPLAN '84 Symp. on Compiler Construction*, 48–57, 1984.

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 465 **COMPILATION OF FUNCTIONAL LANGUAGES BY PROGRAM TRANSFORMATION**
Pascal FRADET, Daniel LE METAYER
28 Pages, Avril 1989.

PI 466 **MODEL BASED DIAGNOSIS : A CASE STUDY IN VIBRATION MECHANICS**
Michèle BASSEVILLE, Albert BENVENISTE
26 Pages, Avril 1989.

PI 467 **DELAI DE COMMUNICATION ENTRE NOEUDS VOISINS SUR L'iPSC/2**
Patrice BURGEVIN, André COUVERT, René PEDRONO
16 Pages, Avril 1989.

PI 468 **OBSERVING GLOBAL STATES OF ASYNCHRONOUS DISTRIBUTED APPLICATIONS**
Jean-Michel HELARY
24 Pages, Avril 1989.

PI 469 **ON-LINE MODEL CHECKING FOR FINITE LINEAR TEMPORAL LOGIC SPECIFICATIONS**
Claude JARD, Thierry JERON
16 Pages, Mai 1989.

PI 470 **PROGRAMMING WITH MALI - THE INTERPRETATION OF PROLOG PROGRAMS**
Louis CHEVALLIER, Serge LE HUITOUZE, Olivier RIDOUX
16 Pages, Mai 1989.

PI 471 **VERS UNE PROBLEMATIQUE DE L'ALGORITHMIQUE REPARTIE**
JeanMichel HELARY, Michel RAYNAL
12 Pages, Mai 1989.

PI 472 **DERIVATION SYSTEMATIQUE D'UN ALGORITHME DE SEGMENTATION D'IMAGES - UN EXEMPLE D'APPLICATION DU FORMALISME GAMMA**
Christian CREVEUIL, Gersan MOGUEROU
46 Pages, Mai 1989.

PI 473 **MICROCODE OPTIMIZATION FOR THE PCS PROCESSOR**
François BODIN, François CHAROT, Charles WAGNER
26 Pages, Mai 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique